NFV-RBCast: Enabling the Network to Offer Reliable and Ordered Broadcast Services

Giovanni Venâncio*, Rogério C. Turchetti†, Elias P. Duarte Jr.*
*Federal University of Paraná, Curitiba, Brazil
†Federal University of Santa Maria, Santa Maria, Brazil
Email: gvsouza@inf.ufpr.br, turchetti@redes.ufsm.br, elias@inf.ufpr.br

***Abstract*—Reliable Broadcast is a classic abstraction for the development of fault-tolerant distributed applications. Informally, reliable broadcast ensures that messages sent to a set of processes are delivered by all correct processes. Moreover, the source may require the delivery of all messages in a particular order. In this case, several different types of orders can be defined, such as the total order, FIFO (First-In First-Out) order, and causal order. In practice, the implementation of reliable and ordered broadcast is not a trivial endeavor. Current solutions are executed by end-users along with their applications. These solutions are often complex to maintain and operate, and require user resources to execute. In this work we propose a strategy to alleviate the user from this burden. NFV-RBCast is a network function that allows the network itself to offer reliable and ordered broadcast services. We employ virtualization technologies to implement the broadcast services using NFV (Network Function Virtualization) technology. NFV-RBCast is based on a sequencer that is executed within the network and establishes message ordering. A communication interface featuring broadcast and delivery primitives is employed by applications to use the broadcast service. A proof-of-concept prototype was implemented and experimental results are reported showing the latency and overhead of the sequencer as well as the throughput for a varying number of processes.**

I. Introduction

Reliable broadcast is a classical building block for the implementation of fault-tolerant distributed applications. Informally, reliable broadcast ensures that messages sent to a set of processes are delivered by all correct processes [1]. Moreover, the source may require all messages to be delivered in a particular order. In this case, there are several different types of orders that can be defined. If all correct processes have to deliver all the messages in exactly the same order, the broadcast is called atomic. If the order is that in which messages were sent by source, the broadcast is called FIFO (First-In First-Out). Causal broadcast determines that the messages are delivered in the order that ensures causal precedence [2].

In [3] the authors argue that the performance of reliable broadcast algorithms is affected by a trade-off between the number of communication steps and the number of messages needed to complete the algorithm. However, in addition to these performance aspects, it is also necessary to choose an algorithm that can be implemented in the particular network environment on which the application will run.

In practice, the implementation of reliable and ordered broadcast is not a trivial endeavor. If the service is offered in the application itself, the application developer has to deal with complex implementation details [4]. Another alternative is to employ specialized middleware, which is also executed and managed by end-users themselves.

In this work we present NFV-RBCast, a novel alternative to implement reliable broadcast that frees the user from having these burdens: the network itself offers broadcast services which are implemented in the network, instead of executing on end-user hosts, either as an application or middleware. NFV-RBCast provides several types of broadcast: reliable broadcast, atomic reliable broadcast, atomic FIFO reliable broadcast, and atomic causal reliable broadcast.

The proposed implementation of broadcast services is based on Network Function Virtualization (NFV) and Software-Defined Networking (SDN). NFV-RBCast takes advantage of virtualization technologies to enable the network itself to provide reliable and ordered broadcast. In fact, the total order of messages is ensured through the use of a sequencer that is also executed within the network. The sequencer is implemented as a Virtualized Network Function (VNF), called VNF-Sequencer which implements a moving sequencer in order to increase availability and performance.

The broadcast and delivery primitives are available through an API accessed by the distributed application, called RBCast. The API provides primitives for broadcast and delivery and is available at end-user hosts. A proof-of-concept prototype was implemented and results are reported showing the broadcast latency and the overhead of the VNF-Sequencer, as well as the throughput while varying the number of processes. In addition, VNF-Sequencer was evaluated in the presence of crashes and overhead.

The rest of this work is organized as follows. Section II describes related work, before presenting basic concepts related to message broadcasting and also the virtualization of network functions. Section III describes the reliable and ordering broadcast algorithms implemented in NFV-RBCast and the VNF-Sequencer. The experiments are presented in Section IV and conclusions follow in Section V.

II. Background and Related Work

In this section an overview of definitions related to both message broadcasting and NFV technology are presented, followed by a description of relevant related work.

978-1-7281-6622-3/19/$31.00 ©2019 IEEE

A. Reliable and Ordered Broadcast

Two processes of a distributed system can communicate by exchanging messages across a communication channel. Reliable broadcast allows more: a process transmits a message that must be reliably delivered by *all* correct processes of the system [1]. Reliable broadcast is a fundamental building block for the development of fault-tolerant distributed applications [5], [6].

Reliable broadcast can be defined in terms of two primitives [7]: *broadcast(m)* and *deliver(m)*, where m is a message.

- The *broadcast(m)* primitive is invoked by the sender to broadcast m to all processes.
- The *deliver(m)* primitive causes the delivery of message m by each correct process to the application that is the actual destination of the message.

Although reliable broadcast ensures message delivery, it does not impose any restriction on the order in which messages are delivered. There are actually several types of broadcast that ensure each a different order that can be applied to the sequence of messages delivered by the processes, as described next.

Atomic broadcast [8] ensures that all correct processes deliver all messages in the same order, called total order. In general, this type of broadcast guarantees the total order for a sequence of messages that is globally agreed upon. In other words, the total order required by atomic broadcast implies that correct processes eventually deliver the same sequence of messages. Note that any sequence will do, as long as all processes deliver all messages in that order.

For some applications, the context of a message depends on other messages previously received from the same source. The broadcast that meets this requirement is called FIFO broadcast, which ensures that messages are delivered by all correct processes in the order they were sent by the source. In the present work the FIFO property was implemented with atomic broadcast, ensuring that total order also follows the FIFO order. We therefore call this type of broadcast as "FIFO atomic broadcast".

Alternatively, a message m may also depend on the messages that the sender of m had delivered before sending m. In this case, the FIFO order is not sufficient: an order that takes into account events with causal precedence is necessary. The concept of causality in the context of distributed systems was formalized by Lamport [2]. If an application requires an order that respects events with causal precedence, it is necessary to use causal broadcast. In this work we implemented "atomic causal broadcast" which is a reliable broadcast that satisfies both causal order and total order.

B. Using a Sequencer to Order Messages

There are several different ways to ensure the total order of messages in a distributed system. A classic and simple approach is to use a *sequencer*, which is a process responsible for receiving all messages and forwarding them to the receivers according to some order. As all correct processes receive all messages from the sequencer, the global total order can be ensured [1].

The process defined to be the sequencer is thus the entity responsible for defining the message order, which are constructed as follows. To broadcast a message m, a source sends m to the sequencer. Upon receiving m, the sequencer assigns a sequence number to m and relays m – with the assigned sequence number – to the destinations, *i.e.* all processes. The correct processes then deliver the messages according to the sequence numbers assigned by the sequencer.

The sequencer itself can be implemented in different ways: it can be fixed, or a moving sequencer, also privilege-based, among other alternatives [1]. In a fixed sequencer algorithm, only one process is responsible for ordering messages on the network. On the other hand, a moving sequencer algorithms allows the role of sequencer to be interchanged between multiple processes. The advantage of this approach is to distribute the load among the processes in addition to tolerate failures. In our solution, the ordering of messages is implemented by a moving sequencer, which is described in more detail in Section III.

C. Network Function Virtualization

Network Function Virtualization (NFV) is an emerging technology that uses software virtualization techniques to implement network functions that have been usually deployed as hardware middleboxes. A Virtual Network Function (VNF) is implemented in software and can be executed on general purpose hardware (*e.g.,* x86 architecture) [9]. Thus, with NFV technology it is possible to design, deploy, and manage network functions in a fraction of the time it often takes to do the same in non-virtualized settings [10].

Typically, a network function (*e.g.,* firewalls, NAT devices, routers) can be started on demand, just as it is needed. In comparison with their traditional equivalents which are usually implemented using proprietary hardware/software, VNFs are easier to manage and operate, and are simply destroyed when they are no longer needed. VNFs can also have their physical resources automatically adjusted, scaling up or down according to the demand, making efficient use of system resources. NFV technology also saves energy and reduces the requirements in terms of physical space [11]. NFV is often deployed together with the Software Defined Network (SDN) technology, with both technologies complementing each other.

D. Related Work

In this subsection we describe some of the most relevant platforms and protocols that provide atomic broadcast primitives and are thus related to NFV-RBCast. The major difference between these strategies and our work is *where* the broadcast algorithms are executed – our solution enables the network itself to offer reliable and ordered broadcast services.

In [12] the authors propose Zab, an atomic broadcast protocol defined in the context of Zookeeper[1]. ZooKeeper provides multiple coordination services to distributed applications. In particular, Zookeeper implements a primary-backup replication strategy in which a primary process executes operations and uses Zab to propagate the corresponding incremental state changes to the backup processes. Zab employs a fixed sequencer, called leader, chosen with leader election algorithm. Each process that performs an atomic broadcast must send the message to the leader. For the delivery of the messages to the other processes, the leader executes an algorithm similar to two-phase commit, where a request is made, votes are collected so that the protocol can commit. To ensure FIFO order, all communications use TCP connections between all pairs of system processes.

In [4], the authors propose a solution called NOPaxos (Network-Ordered Paxos) which relies on the network to establish message ordering that is used to provide strongly consistent replication without executing distributed consensus. The purpose is to to provide an efficient service for consistent replication in data centers. The protocol is only executed when necessary – in particular as packets are dropped – avoiding constant synchronization between processes. A sequencer that is deployed in the network is used to establish the order of messages. This sequencer is implemented directly on the switches. The authors show that this strategy allows the deployment of low-cost replication that presents low latency and high throughput.

ISIS is the classic distributed middleware proposed a couple of decades ago [13] which provides consistency guarantees for distributed processes. The ISIS ABCAST primitive provides atomic broadcast. Furthermore, it offers a wide range of consistency levels for applications by employing the virtual-synchronous model. The system defines groups of processes and the global order of messages is guaranteed even between overlapping groups (*i.e.,* one or more processes in different groups). Since ensuring the total order of messages can be a costly task, the ISIS system also provides weaker ordering primitives in exchange for better performance.

In [14], the authors propose primitives for broadcast communication that were integrated with the early Amoeba distributed operating system. Amoeba was able to provide reliable and ordered broadcast communication for groups of processes. The proposed protocol assumes both communication and process failures. The protocol was defined within the operating system kernel. All processes must run on the same hardware, executing the same kernel and the same application. The protocol was based on a centralized sequencer, which is one of the members of the group.

It is possible to conclude that platforms and protocols of related work mostly run on end-users hosts. Moreover, while some solutions provide a complete set of primitives for message ordering, others only provide partial functionality. In the present work we propose the use of a VNF technology to implement, in the network itself, several primitives of reliable and ordered broadcast. We claim that the complexity for the development of distributed applications is reduced, since reliable and ordered broadcast are provided as services offered by the network itself. Finally, our solution does not require any special hardware or system software modification in order to run the several types of broadcast.

[1]https://zookeeper.apache.org/

III. NFV-RBCast: Architecture and Algorithms

The purpose of NFV-RBCast is to enable the network itself to offer reliable and ordered broadcast services. Distributed applications have access to the services by invoking broadcast primitives. NFV-RBCast assumes reliable communication channels – which are for instance implemented in the Internet with the TCP transport protocol. Several classic algorithms for ensuring the delivery and order of the messages form the NFV-RBCast core. The architecture follows a modular, layered design for service construction *e.g.,* atomic broadcast is built on top of reliable broadcast.

The basic reliable broadcast algorithm that NFV-RBCast implements is the one proposed by Chandra and Toueg [15]. This algorithm ensures the delivery of messages as follows: the source process that starts the broadcast sends message m to all processes. When a process receives m for the first time, it delivers m; furthermore if that process detects that the source has crashed, it sends m to all correct processes. In this way if the sender fails before completely sending the message to all processes, a process that receives the message solves the problem. Thus all correct processes eventually receive and deliver the message. The header of every message m includes a field denoted $local\_seq$, which contains the identity of the sender (*i.e.*, the process identifier) and a field with the local message sequence number. In this way every message can be uniquely identified and the NFV-RBCast can concurrently handle multiple messages coming from multiple sources.

Next we provide details on how message ordering is implemented, and present the NFV-RBCast architecture.

A. Message Ordering

NFV-RBCast employs a sequencer to establish the message ordering. We implemented a moving sequencer which is described in Section III-C. The sequencer orders messages according to two criteria, the first is the local order established at the the source process – which corresponds to the FIFO broadcast order. The other criterion establishes a global order that makes it possible to guarantee the atomic delivery of messages.

Algorithm 1 enforces the atomic delivery of messages by all correct processes. In the algorithm, processes can assume three distinct roles: sender, sequencer, and receiver. The sender (p_i) executes the broadcast primitive for message m, and defines the type of algorithm to be used.

Message m is broadcast with information about the sender identifier, local message counter ($local\_seq$), the type of broadcast algorithm that will be used, and the payload. The global counter is inserted by the sequencer into the message

Algorithm 1 Algorithm to totally order messages.

sender:

Init:
  $RBtype \coloneqq AtomicRB$ {Choose the algorithm}
  local_seq $\coloneqq$ 1
upon broadcast(m) **do**
  broadcast(m, local_seq, $RBtype$)
  local_seq $\coloneqq$ local_seq + 1

sequencer:

Init:
  global_seq $\coloneqq$ 1 {Global counter used to deliver m}
upon receive (m, local_seq, $RBType$) **do**
  broadcast(m, global_seq)
  global_seq $\coloneqq$ global_seq + 1

receiver:

Init:
  nextMsg $\coloneqq$ 1
  pendingMsg $\coloneqq \emptyset$
upon receive (m, global_seq) **do**
  pendingMsg $\coloneqq$ pendingMsg $\cup \{m\}$
  while ($\exists$ ($m' \in$ pendingMsg $\wedge$ global_seq = nextMsg)) **do**
    deliver(m')
    pendingMsg $\coloneqq$ pendingMsg $\setminus \{m'\}$
    nextMsg $\coloneqq$ nextMsg + 1
  end while

and incremented after the message is sent. Each process that receives m adds the message to the list of pending messages ($pendingMsg$). Then a check is run to verify whether message m' with local counter equal to that of the next message ($nextMsg$) is already in the $pendingMsg$ list. Note that $nextMsg$ implicitly identifies the source, so that the sequencer can handle multiple concurrent broadcasts from multiple senders. When the required conditions are satisfied, message m' and all other pending messages that satisfy the selected order are delivered to the application.

In case the application requires not only the atomic order but also the FIFO order, the sequencer implements the delivery according to Algorithm 2. In the FIFO broadcast algorithm, it is necessary to verify that the local counter of message m sent by process p_i, is that of the next message expected. If it is not, the message has to be classified as pending and it must wait to be delivered to guarantee the FIFO order. Thus m is added to the pending list ($F\_pendingMsg$).

For example, if p_i broadcasts $m_3^{p_i}$ and $m_4^{p_i}$ (where for instance $m_3^{p_i}$ is a message sent by process p_i and 3 is the $local\_seq$ number) and the sequencer first receives m_4, then this message will be stored in the $F\_pendingMsg$ list and will remain there until m_3 is received. In this example, when the expected message (m_3) is received, the sequencer forwards m_3 as well as m_4. The sequencer also adds the global counter number before forwarding messages $m_{m'}^s$ and $m_{m''}^s$, where $m' < m''$ (in the example, $m' = m_3$ and $m'' = m_4$). Finally, the correct processes receive the messages from the sequencer and deliver those messages as shown in Algorithm 1 (receiver).

As long as there are pending messages in $pendingMsg$ and the global counter number of a pending message corresponds to that of the next message expected ($nextMsg$), then that message is delivered together with others according to the order being used.

Note that the FIFO order is being used and some process p_j broadcasts message $m_1^{p_j}$ (the first message from p_j), the sequencer forwards this message immediately after its receipt, since $m_1^{p_j}$ is independent of any other message transmitted by any other process.

Algorithm 2 Atomic FIFO order algorithm.

sender:

Init:
  $RBtype \coloneqq AtomicFIFO$
  local_seq $\coloneqq$ 1
upon broadcast(m) **do**
  broadcast(m, local_seq, $RBtype$)
  local_seq $\coloneqq$ local_seq + 1

sequencer:

Init:
  nextMsg $\coloneqq$ 1
  F_pendingMsg $\coloneqq \emptyset$ {List of FIFO pending messages}

upon receive (m, local_seq, $RBType$) **do**
  if ($RBtype$ = $AtomicFIFO$) **then**
    if (local_seq = nextMsg) **then**
      broadcast(m, local_seq)
      nextMsg $\coloneqq$ nextMsg + 1
      while ($\exists$ ($m' \in$ F_pendingMsg $\wedge$ local_seq = nextMsg)) **do**
        broadcast(m', local_seq)
        nextMsg $\coloneqq$ nextMsg + 1
        F_pendingMsg $\coloneqq$ F_pendingMsg $\setminus \{m'\}$
      end while
    else
      F_pendingMsg $\coloneqq$ F_pendingMsg $\cup \{m'\}$
    end if
  end if

Hadzilacos and Toueg [16] describe the causal order as a

generalization of the FIFO Order. In fact, it is possible to transform the FIFO broadcast into the Causal Broadcast by proving that the algorithm satisfies both the FIFO order and local order. That is exactly how NFV-RBCast provides causal atomic broadcast.

B. NFV-RBCast Architecture

The NFV-RBCast architecture is shown in Fig. 1. The system is assumed to run in an SDN network. A key component of NFV-RBCast is the sequencer, which is implemented as a VNF (called VNF-Sequencer). Module RBCast offers an API that consists of a set of primitives so that applications can execute the broadcast services. This module is run on the end-user hosts, at the end-points of the communication channel, the hosts running applications that use the broadcast services.

In this work, the VNF-Sequencer is implemented taking advantage of SDN network. SDN technology has several facilities to deploy policy-based message routing. Note however that using SDN is not a strict requirement of NFV-RBCast, as the VNF-Sequencer can be easily adapted to run on a traditional network without major changes. As it is, NFV-RBCast relies on an OpenFlow SDN Controller to create the rules for the communication between the distributed applications that make use of the broadcast services and the sequencer. When a process sends a message using reliable and atomic broadcast, the total order is constructed in two steps described next, Process-to-Sequencer and Sequencer-to-Process.

- **Process-to-Sequencer**: Once the application defines the order type to be applied to the messages, the source broadcast the message on the network and the messages are forwarded to an SDN switch from the RBCast interface. Note that in the case of reliable (not atomic) broadcast, it is not necessary to use the sequencer and the messages are delivered after they are received. Otherwise, if the broadcast is ordered, the messages follow the standard operation of the OpenFlow protocol [17]: every message that does not have an entry in the switch flow table (*i.e. packet-in*) is forwarded to the controller. For each *packet-in* received, the controller checks whether the message is an ordered broadcast message, in case it is, the next action is to install an OpenFlow rule in SDN switches to forward all similar messages coming from the RBCast to the VNF-Sequencer. For the next messages of the same flow no additional rules need to be created: the messages are directly sent to the VNF-Sequencer. This strategy has an advantage as the source does not need to know the IP address of the sequencer or even that a sequencer is being used.

- **Sequencer-to-Process**: When message m is received by the VNF-Sequencer, it selects the type of broadcast defined in the Process-to-Sequencer step, and executes the corresponding broadcast algorithm (described in Section II-B). When m is forwarded from the VNF-Sequencer to the destinations, it receives a global sequence value assigned by the sequencer. Message m is sent following the same procedure described in the Process-to-Sequencer step. If m does not have a corresponding rule in the flow table of the SDN switch, that message is forwarded to the SDN controller which installs a OpenFlow rule to forward all traffic from the VNF-Sequencer to all destination processes. Finally, the processes deliver m taking into account the sequence number determined by the VNF-Sequencer.

An alternative to implement the sequencer as a VNF would be to do that directly in the SDN controller. However, we do not employ this strategy for two reasons. The first is for performance issues, since adding ordered broadcast modules to the controller would result on extra load (*i.e.*, packet processing and message ordering), possibly compromising its performance and availability. Moreover, executing the sequencer as a VNF reduces the complexity of development and management, since implementing and deploying broadcast algorithms in the controller implies in several modifications of the controller itself.

Fig. 2 shows the cost of one execution of the broadcast algorithm using the sequencer strategy. The example is based on the model proposed in [18]. In the figure, there is two processes P1 and P2. The transmission of a message from a sending process P1 to a destination process P2 requires two types of resources: CPU and network. The VNF-Sequencer is placed inside the network. When message m is transmitted in the network, the VNF-Sequencer selects the type of broadcast defined by the sender and executes the corresponding broadcast algorithm forwarding m to the destinations. The CPU resources represent the processing produced by the communication layers, throughout the emission and the reception of a message. Each process has assigned one CPU. The transmission end-to-end delay corresponds to the following steps:

1) *Process sends or receives a message:* m leaves or enters the CPU resource. In this latter, m waits for CPU to be available.
2) *Message is performed by the CPU resource:* m uses the resources CPU for λ time units.
3) *Message enters the network or in a queue:* m waits until the network or CPU is available.
4) *Message is performed by the network:* m uses the network resource for some time (*e.g.,* 1-time unit).

The measure of the cost of one execution of the broadcast algorithm is given by $(4\lambda + 2)$. This cost denotes that the sequencer is a good strategy as illustrated in the studies performed by the authors [18].

C. VNF-Sequencer

There are several ways to implement a sequencer, which can be a fixed, moving, privileged-based, among other alternatives [1]. Although the fixed sequencer is the simplest to implement and manage, a major drawback is that it becomes a single point of failure, affecting availability and performance, since

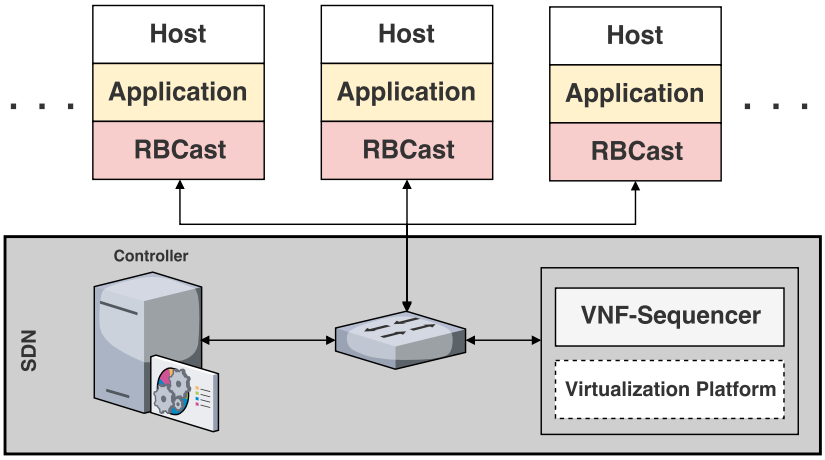

Fig. 1. NFV-RBCast architecture.

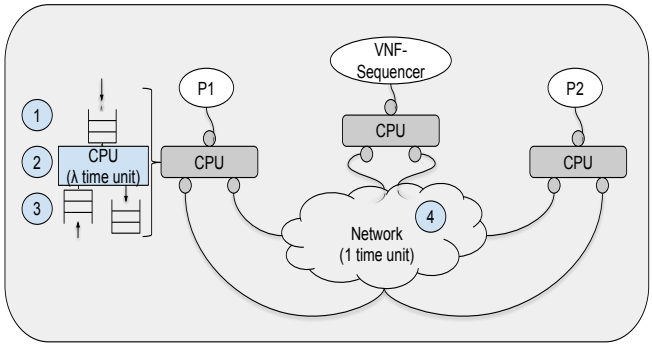

Fig. 2. End-to-end delay to broadcast a message.

only a single node is responsible for the task. As the load on the fixed sequencer increases, there can also be an impact on throughput and latency. In order to avoid this problems, the VNF-Sequencer was implemented as a moving sequencer. The main advantages of this approach is twofold: *(i)* it tolerates sequencer crashes and thus increases the availability and; *(ii)* the load can be balanced among sequencer nodes, improving the performance.

Fig. 3 shows the VNF-Sequencer architecture. As it is a moving sequencer, it is deployed on a pool of nodes, each of which executes a VNF instance. Each VNF instance can assume the role of sequencer and all instances execute the same algorithms described in Section III-A. The number of VNFs is customizable, according to the required availability and performance levels. We note that using NFV technology makes it easier to change the size of the pool.

At any time instant, only one of the VNF instances performs the role of sequencer. A token is employed, so that the instance that holds the token is responsible for messages ordering. The token consists of a message (*node_id*, *counter*) where *node_id* is the node identifier and *counter* corresponds to the latest value of the global message counter. The *counter* consists both of the global sequence number employed by the atomic broadcast algorithm and the local sequence number for each sender used to enforce the order of the FIFO broadcast algorithm. In Fig. 3, the sender is node S1 that sends messages to the VNF-Sequencer. The token is hold by Node 2, which establishes message ordering, and sends the ordered messages to the receivers.

The VNF-Sequencer contains two additional modules: a Monitor Module (MM) and a Token Manager Module (TMM). The MM implements two components, a Failure Detector (FD) based on the heartbeat strategy and a performance monitor (PM). The TMM also implements two components, a load balancer and an election module. Each component is described below.

1) Failure Monitoring: Unreliable failure detectors were proposed by Chandra and Toueg [15] as abstractions that, depending of their properties, can be used to solve consensus in asynchronous systems with crash faults. A failure detector is defined as an "oracle" that can be accessed by a node to obtain information about the state of the other nodes of the distributed system. Failure detectors are said to be *unreliable* as they can make mistakes, *i.e.* report an incorrect state. For example, a monitored node can be incorrectly suspected to have crashed, but later the suspicion can be raised if the the failure detector learns that the node is alive.

A common approach to implement failure detectors is to monitor the message exchange pattern. As mentioned before, the strategy employed by our FD is classified as a heartbeat-based. Using a heartbeat FD, the monitored nodes (*i.e.,* moving sequencer pool) sends heartbeat messages at periodic time intervals to the monitor. Based on the observed message arrival pattern, the failure detector computes a timeout interval. If a message is not received by the monitor within this timeout interval, the monitored node is suspected to have crashed. If the sequencer is suspected, the TMM elects a new sequencer.

2) Performance Monitoring: In addition to monitoring failures, the PM (Performance Monitor) module of the MM monitors the performance of the pool of sequencer nodes. For this purpose, the PM periodically collects metrics from each

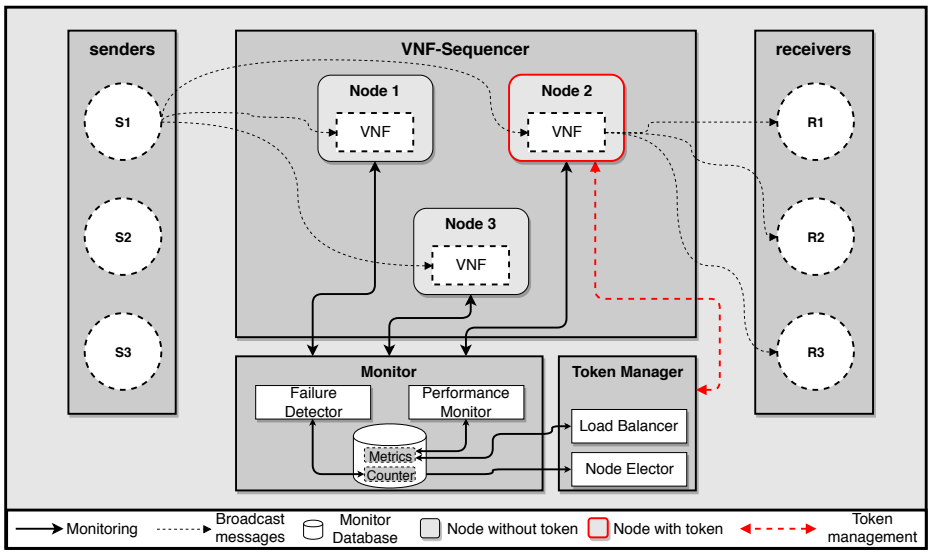

Fig. 3. The VNF-Sequencer architecture.

node and stores this information in a local database. Metrics include CPU, memory, and bandwidth usage, among others.

With these metrics, the TMM can prevent network performance degradation by identifying overloaded nodes and performs load balancing among the pool nodes. Multiple node-selection policies can be applied in this context. For the experiments reported in the next section, the policy used verifies if the CPU usage is above a predefined threshold (*e.g.,* above 90%). Based on the collected metrics, if the values for the node that is currently serving as sequencer matches the threshold, the TMM will elect a new sequencer.

3) Token Manager Module: The TMM (Token Manager Module) is responsible for managing the token and for the election of sequencers. Note that a sequencer in charge maintains a global message counter that increases monotonically. As another sequencer is elected, the TMM has to ensure that it receives the correct, up-to-date global message counter. In order to ensure the smooth transition from one sequencer to another, the counter has to be properly monitored.

As described previously, the FD periodically monitors each node using heartbeat messages. To minimize the number of messages, the token synchronization task – which consists of obtaining the updated global message counter from the sequencer in charge can be *piggybacked* on each heartbeat message.

Two steps are required to change the sequencer from a node to another. The first step consists of the decision itself that a new sequencer has to be elected. The second step consists of the election itself. The two steps are described below.

Deciding on a new election. The decision that a new sequencer has to be elected starts with the TMM detecting that the sequencer in charge node is either overloaded or has crashed. Monitoring resource consumption can be used to assess that a sequencer is overloaded. In this case, the TMM sends a message to the sequencer currently in charge requesting the global message counter and, at the same time, informs the node that it no longer has the token. In the case in which the FD detects that the sequencer has crashed, the TMM can not obtain the updated global counter, furthermore the token itself is lost and messages may have been lost for a period of time. In this case, the TMM performs a rollback to the last received counter, which it stores locally. In both situations, after detecting that it is necessary to change the sequencer, the election begins.

Electing a new sequencer. To elect a new sequencer, the TMM selects among the nodes in the moving sequencer pool the one that has presented the lowest CPU utilization. In order to avoid making a poor decision due to utilization peaks, an average of the latest 10 utilization samples is employed. Once the elected node is decided, the TMM broadcasts the token to the moving sequencer pool containing the identifier of the new sequencer (*node_id*) and the updated global message counter (*global_seq*). All nodes receive this message and check if the *node_id* in the token is equal to its own *node_id*. If it is, the node assumes the role of sequencer in charge.

The next section describes a proof-of-concept prototype and experimental results executed for evaluating the performance of VNF-Sequencer in several scenarios.

IV. Experimental Evaluation

In this section we report the results of experiments executed in order to evaluate the performance of proposed in-network broadcast services. The experiments were executed on a proof-of-concept prototype, implemented with the Ryu [19] SDN controller, Open vSwitch [20], and using OpenFlow 1.3 for the communication between the switch and the controller. The prototype was executed on a physical machine based on an Intel Core i5-7200U@2.50GHz processor with 4 cores, 8 GB RAM and running Ubuntu 18.04. We developed a client application responsible for generating data flows. This appli-

cation broadcasts messages continuously. Each node of the VNF-Sequencer pool and the Ryu controller are deployed on containers running on the Docker platform [21]. Experiments are presented to evaluate the overhead and the throughput of the VNF-Sequencer.

A. Evaluation of VNF-Sequencer Overhead

In Section III, we mentioned that the purpose of the VNF-Sequencer is to guarantee the atomic delivery of messages and that it is implemented within the SDN network. Using a sequencer is actually the most efficient way to guarantee the total order of messages. On the other hand, there is a cost in terms of the overhead as the sequencer has to process *all* messages that are sent through the ordered broadcast service. In this way, the first experiment aims to measure the impact of the sequencer on the latency of atomic broadcast which is compared with reliable broadcast, which does not use a sequencer.

In the experiment shown in Fig. 4, the topology consists of a switch, an SDN controller and a varying number of processes that take part of the reliable broadcast instance. One of the participating processes implements a client application responsible for broadcasting messages that are initially forwarded to the local RBCast interface. When other processes receive the message from their RBCast interfaces, messages are delivered to the application. In this experiment, we measured the message delivery latency which corresponds to the time elapsed from the moment the message was transmitted by reliable broadcast until the instant that all processes have completely delivered the message to the application. In this experiment, each value shows the mean of 10 samples and each sample consists of 1000 broadcasts of 1KB messages.

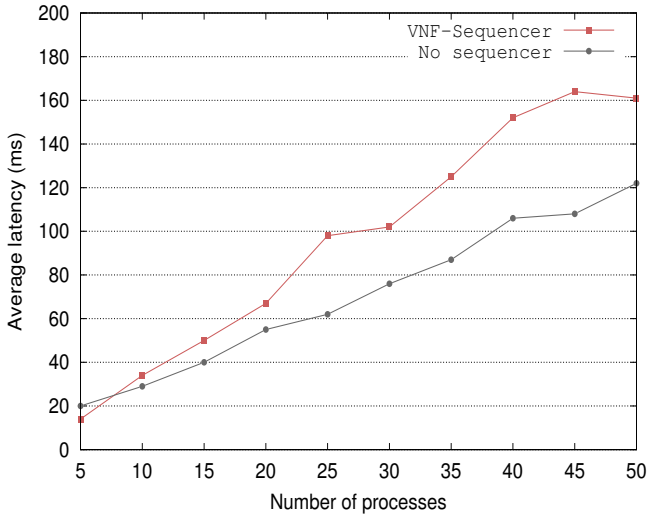

Fig. 4. Comparison of the latency for message delivery.

Fig. 4 shows the results of the comparison of the communication with (*VNF-Sequencer* curve) and without a sequencer (*No Sequencer* curve). As expected, we can note that in most cases the latency is higher with the use of the VNF-Sequencer. In quantitative terms, we observed that the average latency when the sequencer is used is approximately 32.1% higher than without the sequencer. It is also possible to notice that the latency increases in both cases as the number of processes grows. Up to 20 processes, the latency increases by about 20%. It is possible to conclude that in a system with up to 20 processes, the cost to guarantee the total order of messages with the sequencer is a fair price to pay. Above 20 processes, the latency overhead can increase from 34% til up to 56%.

We also evaluated the influence of the message size on the performance. In this experiment, we measured the latency for 50 processes that broadcast messages with sizes varying from 1KB to 16KB. Fig. 5 shows the mean taken from 10 samples each of which consisting of 1000 broadcasts.

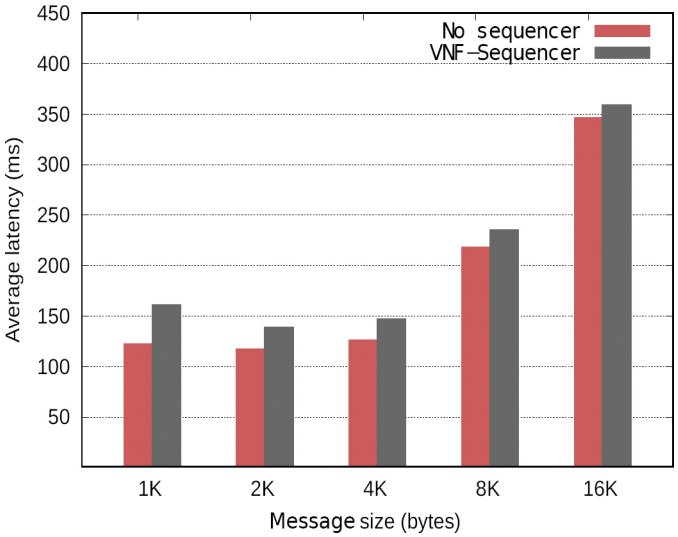

Fig. 5. Comparison of the latency for 50 processes for different message sizes.

Considering these results, the average latency increases by 15.7% with the sequencer. It is also important to note that the latency decreases as the message size increases. For 1KB messages, the latency overhead is about 31.7% larger with VNF-Sequencer. On the other hand, for 16KB messages, the difference drops to only 3.8%. Therefore, it is possible to conclude that as the message size increases the impact of the sequencer on the latency diminishes. It is also possible to observe that without the VNF-Sequencer the latency increases by 224ms, whereas with the VNF-Sequencer it increases by 198ms, *i.e.* it is just 11.6% lower. In other words, as we increase the size of the messages, the rate in which the latency grows reduces when the VNF-Sequencer is employed.

It is important to remember that the VNF-Sequencer introduces an overhead in terms of latency, but this is expected as it is the component that establishes the total order required for atomic broadcast.

B. Evaluation of the Throughput

The next experiment was executed with the purpose of evaluating the throughput of the VNF-Sequencer, varying

the number of processes performing the broadcast. In this experiment, each execution lasts three minutes and results are the average of three executions. The average number of messages handled per second by the VNF-Sequencer is computed. In this way, we measured the maximum number of messages that were broadcast. Up to 50 processes were evaluated, one of which is the sender. The algorithm used in this experiment was the atomic broadcast.

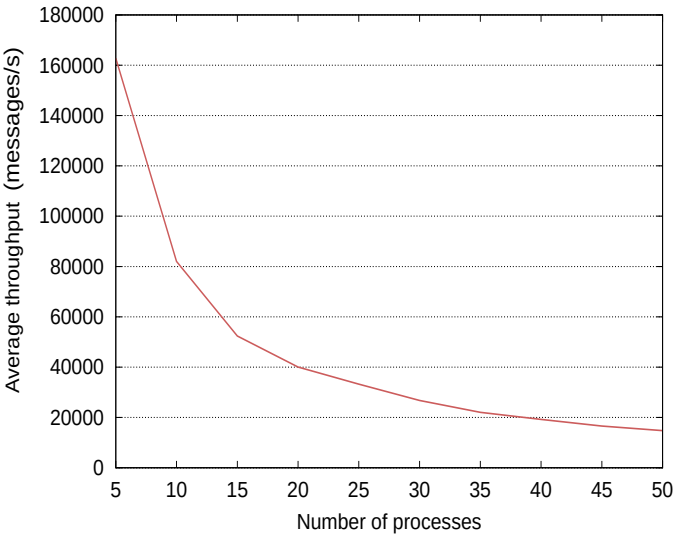

Fig. 6. VNF-Sequencer throughput as the number of processes increases.

In Fig. 6 we show the variation of the throughput as the number of processes increases. In this scenario the processes are running multiple broadcasts. Since the sender process sends messages to VNF-Sequencer at its maximum rate, the throughput of the VNF-Sequencer is determined by the time required to broadcast a message to all destination processes. Thus, the time to broadcast a message to a small number of processes is less than the time to do that to a larger group of processes. As a result, the throughput decreases as we increase the number of processes participating in the broadcast. In particular, we highlight that for up to 50 processes, the VNF-Sequencer was never the bottleneck, reaching a throughput of approximately 15000 messages per second.

C. Throughput with Failures and Overhead

As seen in Section III-C, the VNF-Sequencer is a moving sequencer which raises the availability and performance of the solution in comparison with the simpler fixed sequencer. In this experiment we evaluate the VNF-Sequencer throughput in the presence of crash faults as well as in the presence of overloaded sequencer nodes.

Fig. 7 shows the VNF-Sequencer throughput for one minute of execution (x-axis). In this experiment, a client continually sends atomic broadcast messages. The moving sequencer pool has been configured with 3 nodes while 1 VNF instance runs on each one. Moreover, the system consists of 20 processes that deliver the broadcast messages.

At time instants 20 and 40, either a crash fault was injected (*VNF-Sequencer with Failures* curve) or the sequencer became overloaded (*VNF-Sequencer with Overhead* curve). The crash fault was injected by deleting the container hosting the sequencer in charge. Analogously, in order to cause a significant overhead on the sequencer, we spawn multiple jobs that consist of a sequence of operations that are CPU-intensive.

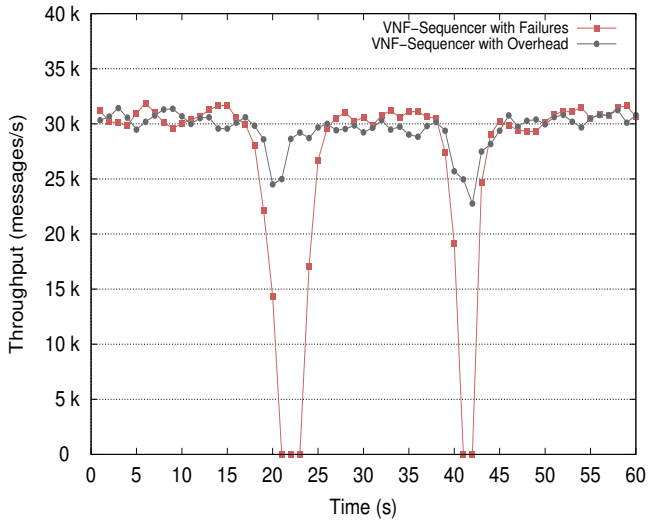

Fig. 7. VNF-Sequencer throughput under crashes and an overloaded sequencer.

After a process crashes, it is possible to note that the VNF-Sequencer throughput is reduced to zero for an interval of 2 to 3 seconds. This is the time interval from the instant the process has crashed to the instant at which the FD starts to suspect that process. Meanwhile, the FD performs the rollback to the last global message counter, and elects a new sequencer.

As the overhead of the sequencer increases, the throughput only decreases, instead of simply going down to zero as in the case of crashes. The recovery interval is also in 2 to 3 seconds range, which corresponds to the time required by the TMM to detect the situation, after which it obtains the updated global message counter, and elects a new sequencer. In this case, the TMM does not perform any rollback operation, since no messages were lost.

These experimental results allow the conclusion that NFV-RBCast is both effective – by enabling the network itself to offer reliable and ordered broadcast services – and robust – as it ensures the correct delivery of messages even when after sequencer crashes or becomes overloaded.

V. Conclusion

A very large number of distributed applications employ reliable and ordered broadcast services. Systems that implement those services are usually deployed on end-hosts, either in the application or as underlying middleware. In this work we presented a different alternative: we employ NFV technology to enable the network itself to provide reliable and ordered broadcast services. Thus applications can simply

invoke the service available from the network, and users not need to install/maintain/run the services on their own end-hosts. NFV-RBCast provides several types of broadcast services to ensure reliable and ordered delivery of messages to distributed applications: reliable broadcast, atomic reliable broadcast, atomic FIFO reliable broadcast, and atomic causal reliable broadcast. The system is based on a moving sequencer which is implemented as a VNF. The VNF-Sequencer manages message transmissions and establishes the required ordering of the messages. Moreover, the strategy used to implement the VNF-Sequencer tolerates failures and allows load balancing among the pool of sequencer nodes. In addition, the VNF-Sequencer features other modules that perform several other tasks, such as reverting the system to a previous state (roll-back).

A proof-of-concept prototype was implemented and results show that the VNF-Sequencer meets the performance expectations. The cost in terms of the latency and overhead was presented for different scenarios, both varying the number of broadcast participants and the size of the messages transmitted. Furthermore, the throughput was measured by increasing the number of processes. In the experiment, it was observed that for up to 50 processes the sequencer did not become a bottleneck. Finally, the VNF-Sequencer was evaluated in terms of the overhead it represents, including in the presence of failures. The VNF-Sequencer was able to mitigate the consequences of node failures as well as of overloaded nodes within an interval of 2 to 3 seconds.

Future work includes the adaptation of NFV-RBCast to the NFV-MANO reference architecture (NFV Management and Orchestration)[2]. Besides being a standard, NFV-MANO would also improve the scalability of the VNF-Sequencer: the size of the moving sequencer pool could be autonomically adjusted to adapt to the current requirements and conditions. Yet another alternative is to investigate effective strategies to implement the total order of messages in a distributed fashion using a consensus algorithm.

REFERENCES

[1] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] R. Ekwall and A. Schiper, "Modeling and validating the performance of atomic broadcast algorithms in high latency networks," in *13th International Euro-Par Conference*, 2007.

[2]https://www.etsi.org

[4] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say no to paxos overhead: Replacing consensus with network ordering," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[5] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 3, pp. 251–273, 1984.

[6] F. Pedone and A. Schiper, "Optimistic atomic broadcast: a pragmatic viewpoint," *Theoretical Computer Science (Elsevier)*, vol. 291, no. 1, pp. 79–101, 2003.

[7] D. Jeanneau, L. A. Rodrigues, L. Arantes, and E. P. Duarte, "An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detector," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, 2016.

[8] D. Cason, P. J. Marandi, L. E. Buzato, and F. Pedone, "Chasing the tail of atomic broadcast protocols," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, 2015.

[9] D. Cotroneo, L. De Simone, A. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network function virtualization: Challenges and directions for reliability assurance," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 37–42.

[10] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, 2016.

[11] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 90–97, 2015.

[12] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. ACM, 2008, p. 2.

[13] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, 1987.

[14] M. F. Kaashoek and A. S. Tanenbaum, "Group communication in the amoeba distributed operating system," in *Distributed Computing Systems, 1991., 11th International Conference on*. IEEE, 1991, pp. 222–230.

[15] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, no. 2, 1996.

[16] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Tech. Rep., 1994.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[18] P. Urban, X. Defago, and A. Schiper, "Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms," in *Proceedings Ninth International Conference on Computer Communications and Networks (Cat.No.00EX440)*, 2000, pp. 582–589.

[19] S. Ryu, "Framework," 2016. [Online]. Available: https://osrg.github.io/ryu/

[20] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch." in *NSDI*, vol. 15, 2015, pp. 117–130.

[21] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.